Aydie's Avenue

Aditya Dinesh K
aydie.in
business@aydie.in
+91 9036 4694 92
Mangalore, Karnataka, India

Product Of Aydie's Avenue

NYC Taxi Demand Prediction

7th April, 2025

Source: aydie.in/NYC-Taxi-Prediction

1. Objective

This project aims to build a machine learning model capable of accurately predicting the demand for yellow taxis in New York City based on historical trip data. By understanding patterns in pickup times, locations, and other relevant features, the model aims to help optimise taxi fleet distribution and improve passenger experience.

1.1 Background

New York City's yellow taxis are an essential mode of transport for millions of residents and tourists. The dynamic nature of urban mobility makes it challenging to maintain optimal taxi availability at the right place and time. Predictive modelling of taxi demand enables better decision-making for:

- Dispatch systems
- Surge pricing mechanisms
- Urban traffic management

This project uses publicly available NYC Yellow Taxi Trip data to analyse historical trends and train machine learning models to forecast future demand at hourly intervals across different city zones.

1.2 Key Features of the Project:

· **Large-Scale Dataset** with millions of trip records including timestamps, geolocations, and fare details.

· **Feature Engineering** to extract meaningful variables such as hour-of-day, day-of-week, location clusters, and weather influence (optional).

· **Modelling techniques** involving time series analysis and supervised learning models.

· **Real-World Deployment Readiness** through the integration of MLOps practices, enabling automated retraining, monitoring, and deployment of the model via cloud platforms.

1.3 Deliverables:

· Cleaned and structured the dataset ready for ML tasks.

· A trained and evaluated predictive model for hourly taxi demand.

· Visualisations and dashboards showing demand trends.

· Deployment-ready pipeline with automated retraining.

· Complete project documentation and source code.

1.4 Tools & Technologies:

· **Languages:** Python

· **Libraries:** Polars, Pandas, NumPy, Scikit-learn, XGBoost, Matplotlib, Seaborn

· **Infrastructure:** Google Cloud Platform / AWS (for MLOps)

· **Version Control:** GitHub

· **Notebook Environment:** Jupyter Notebooks / Google Colab

2. Data Collection

The dataset used in this project is the **NYC Yellow Taxi Trip Data**, publicly available on Kaggle. It contains detailed trip records of yellow taxis in New York City, collected by the NYC Taxi and Limousine Commission (TLC).

Link: https://www.kaggle.com/datasets/elemento/nyc-yellow-taxi-trip-data

2.1 Description of the Dataset

The dataset consists of millions of rows, where each row represents an individual taxi trip. Key attributes in the dataset include:

• pickup_datetime: Date and time of the trip start

• dropoff_datetime: Date and time of the trip end

• pickup_longitude and pickup_latitude: Pickup location coordinates

• dropoff_longitude and dropoff_latitude: Dropoff location coordinates

• passenger_count: Number of passengers during the trip

• trip_distance: Distance traveled in miles

• fare_amount: Cost of the ride

• payment_type: Method of payment

• tip_amount: Tip paid by the passenger

These features provide valuable information for understanding when and where demand peaks across the city.

2.2 Data Acquisition Process

• The dataset was downloaded from Kaggle as CSV files, each file representing one month of taxi trips.

• Files were merged and stored locally for preprocessing and analysis using Python.

• For handling large data, operations were performed using batch loading, filtering based on dates or areas, and leveraging efficient libraries such as **Dask** and **Polars** when required.

2.3 Time Frame

The dataset spans multiple years. For this project, a specific time window was chosen (e.g., January 2016 to March 2016) to maintain focus and reduce computational overhead. However, the approach is scalable to handle more extended periods.

2.4 Storage & Versioning

• Data is stored in a structured folder hierarchy for raw and processed datasets.

• GitHub and Google Drive are used for version control and backups of scripts and smaller sample files.

• Cloud storage (e.g., GCS/AWS S3) is planned for integration in the MLOps phase for scalable processing and deployment.

Data Exploration and Preprocessing

I am loading data into a Polars DataFrame using the month object while converting tip_amount and tolls_amount to the Polars Float64 type during the load. I assign total_records with the total record count of the dataset. [35.5 million records – Jan, Feb, Mar 2016]

```
import polars as pl

schema_overrides = {
    'tip_amount': pl.Float64,
    'tolls_amount': pl.Float64
}

month = pl.read_csv('DataSets/yellow_tripdata_2016-01.csv', schema_overrides = schema_overrides)
total_records = month.shape[0]
```

Executed at 2025.04.07 11:52:33 in 3s 684ms

3.1.2 Performance Metrics:

Mean Absolute Percentage Error (MAPE)

$$\text{MAPE} = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{P_t - \hat{P}_t}{P_t}\right| \times 100$$

Mean Squared Error (MSE)

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(P_t - \hat{P}_t)^2$$

P_t = actual value

$\hat{P}_t$ = predicted value

n = total number of observations

Data Cleaning

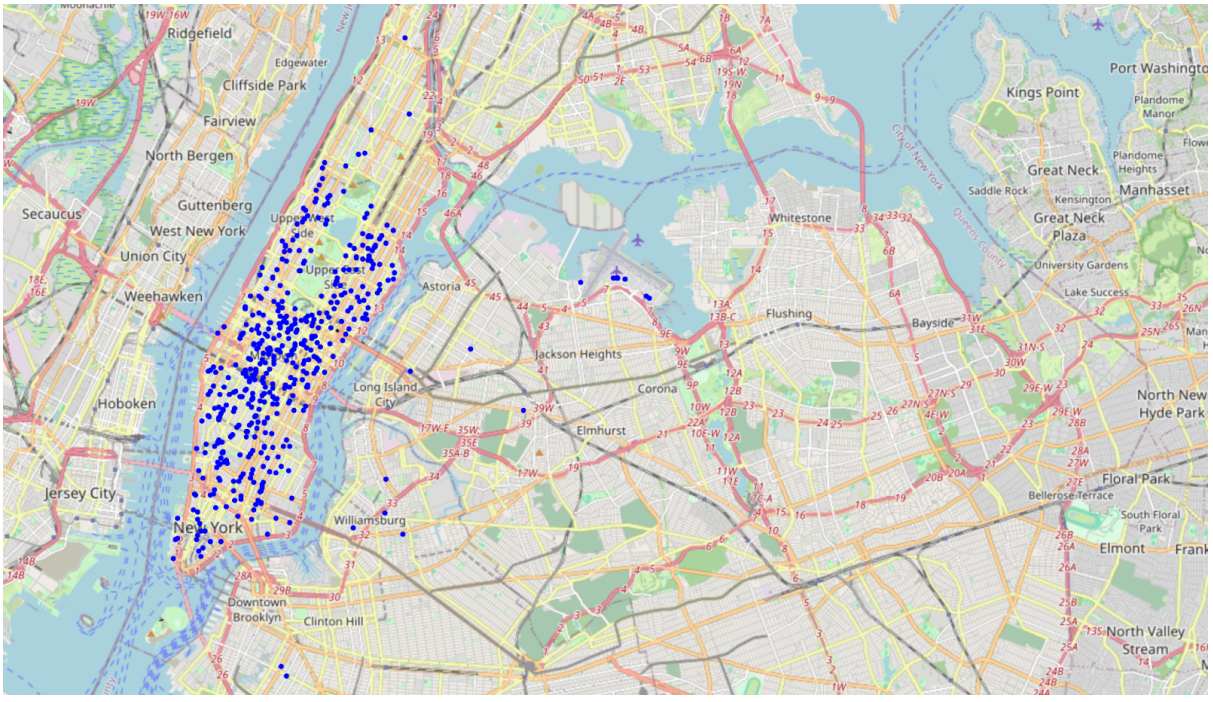
Visualise the outliers using a map [folium library].

Pickup Latitude and Longitude

The approximate bounding box (latitude and longitude) of New York City (NYC) is:

- Southwest Corner (Min Latitude, Min Longitude): (40.4774, -74.2591)
- Northeast Corner (Max Latitude, Max Longitude): (40.9176, -73.7004)

```
sampled_df["is_outlier"] = ~((sampled_df["pickup_latitude"].between(min_lat, max_lat)) &
                             (sampled_df["pickup_longitude"].between(min_lon, max_lon)))

# Create NYC map
nyc_map = folium.Map(location=[40.7128, -74.0060], zoom_start=12)

# Add points to the map
for _, row in sampled_df.iterrows():
    color = "red" if row["is_outlier"] else "blue"
    folium.CircleMarker(
        location=[row["pickup_latitude"], row["pickup_longitude"]],
        radius=1,
        color=color,
        fill=True,
        fill_color=color,
        fill_opacity=0.5
    ).add_to(nyc_map)
```



Calculating Trip Times, Trip Distance and Speed

- Using the datetime module, we calculate trip_distance, trip_time, and speed, and assign them as new dimensions to the monthly DataFrame.

```python
# Convert timestamps into Unix Time
def convert_to_unix(s: str) -> float:
    return datetime.datetime.strptime(s, "%Y-%m-%d %H:%M:%S").timestamp()

# Function to calculate trip times and speeds
def return_with_trip_times(month):
    month = month.with_columns([
        pl.col("tpep_pickup_datetime").map_elements(convert_to_unix, return_dtype=pl.Float64).alias("pickup_unix"),
        pl.col("tpep_dropoff_datetime").map_elements(convert_to_unix, return_dtype=pl.Float64).alias("dropoff_unix")
    ])

    # Compute trip duration in minutes
    month = month.with_columns(
        ((pl.col("dropoff_unix") - pl.col("pickup_unix")) / 60).alias("trip_times")
    )

    # Avoid division by zero: Replace zero trip_times with NaN before calculating Speed
    month = month.with_columns(
        pl.when(pl.col("trip_times") == 0)
          .then(None)
          .otherwise(pl.col("trip_times"))
          .alias("trip_times")
    )

    # Compute speed (handling division by zero)
    month = month.with_columns(
        ((60 * pl.col("trip_distance").cast(pl.Float64)) / pl.col("trip_times")).alias("Speed")
    )

    return month
```

- Remove trip_time < 0, null values and infinite values of speed.

```python
# Check few records
month_with_durations.filter(
    (pl.col("trip_times") > 0) &
    (~pl.col("speed").is_null()) &
    (~(pl.col("speed") == float("inf")))
).head()
```

- **So, we remove and clean the data for all January, February, and March records across different dimensions and attributes.**

Total Data Loss on all outlier removal (Jan):

```
Records Loss On Outliers Removal
Distance: 22661
Speed: 22067
Fare Amount: 29130
NYC Region Lat vs Long:  210900

Total Loss: 284758 out of 10906858
Percentage Retained: 97.389 Percentage Lost:  2.611
```

Unix Time Binning

Convert Time Stamp to Unix Time:

```
# Convert timestamps into Unix Time
def convert_to_unix(s: str) -> float:
    return datetime.datetime.strptime(s, "%Y-%m-%d %H:%M:%S").timestamp()

# Function to calculate trip times and speeds
def return_with_trip_times(month_jan_2016):
    month_jan_2016 = month_jan_2016.with_columns([
        pl.col("tpep_pickup_datetime").map_elements(convert_to_unix, return_dtype=pl.Float64).alias("pickup_unix"),
        pl.col("tpep_dropoff_datetime").map_elements(convert_to_unix, return_dtype=pl.Float64).alias("dropoff_unix")
    ])

    # Compute trip duration in minutes
    month_jan_2016 = month_jan_2016.with_columns(
        ((pl.col("dropoff_unix") - pl.col("pickup_unix")) / 60).alias("trip_times")
    )

    # Avoid division by zero: Replace zero trip_times with NaN before calculating Speed
    month_jan_2016 = month_jan_2016.with_columns(
        pl.when(pl.col("trip_times") == 0)
          .then(None)
          .otherwise(pl.col("trip_times"))
          .alias("trip_times")
    )

    # Compute speed (handling division by zero)
    month_jan_2016 = month_jan_2016.with_columns(
        ((60 * pl.col("trip_distance").cast(pl.Float64)) / pl.col("trip_times")).alias("speed")
    )

    return month_jan_2016
```

Unix Time Binning:

```
def bin_unix_time(df: pl.DataFrame, column_name: str, bin_size_seconds: int) -> pl.DataFrame:
    return df.with_columns([
        (pl.col(column_name) // bin_size_seconds * bin_size_seconds).alias(f'{column_name}_binned')
    ])

# Apply time binning
month_jan_2016_modified = bin_unix_time(month_jan_2016_modified, 'pickup_unix', 600)

# Convert binned seconds to datetime (in milliseconds)
month_jan_2016_modified = month_jan_2016_modified.with_columns(
    (pl.col("pickup_unix_binned") * 1000)
    .cast(pl.Datetime("ms"))
    .alias("pickup_binned_datetime")
)
```

Smoothing Time Series (Winzorisation) of Time Column:

```
# Calculate lower and upper thresholds for "trip_times" (e.g., 1st and 99th percentile)
lower_threshold = month_jan_2016_modified.select(
    pl.col("trip_times").quantile(0.01).alias("lower_threshold")
).to_series()[0]

upper_threshold = month_jan_2016_modified.select(
    pl.col("trip_times").quantile(0.99).alias("upper_threshold")
).to_series()[0]

print("Lower threshold:", lower_threshold)
print("Upper threshold:", upper_threshold)

# Cap the outliers in "trip_times" by winsorizing, and store the result in a new column "trip_times_capped"
month_jan_2016_modified = month_jan_2016_modified.with_columns(
    pl.when(pl.col("trip_times") < lower_threshold)
      .then(lower_threshold)
      .when(pl.col("trip_times") > upper_threshold)
      .then(upper_threshold)
      .otherwise(pl.col("trip_times"))
      .alias("trip_times_capped")
)

# Optional: Display a few rows to verify the changes
print(month_jan_2016_modified.select(["trip_times", "trip_times_capped"]).head())
```

*Do the same above steps for Feb, March dataset and export the data to CSV file.

Data Preparation / Clustering

Merge Jan, Feb, March data row-wise

```
# Concatenate row-wise
df_all = pl.concat([df_jan, df_feb, df_mar], how="vertical")
```

Clustering Of Data

```
import numpy as np
from sklearn.cluster import MiniBatchKMeans
import gpxpy.geo # for haversine distance

# Getting coordinates from Polars DataFrame
coords = df_all.select([
    'pickup_latitude', 'pickup_longitude'
]).to_numpy()

# Will store nice cluster counts for each cluster size
neighbours = []
```

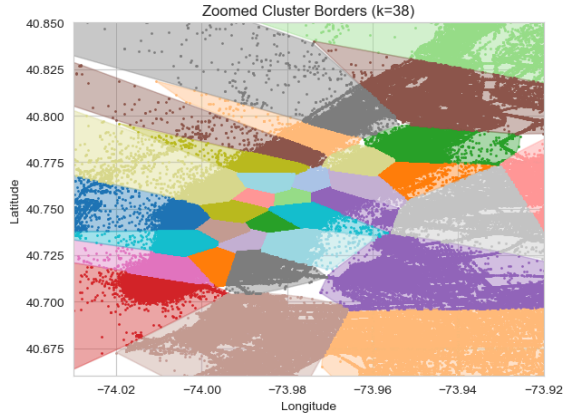
```
# Function to compute how many clusters are close to each other
def find_min_distance(cluster_centers, cluster_len):
    less2 = []        # Stores count of close clusters for each center
    more2 = []        # Stores count of far clusters for each center
    min_dist = 10000  # Initialize with a large distance

    for i in range(cluster_len): # For each cluster center
        nice_points = 0  # Clusters with at least one nearby cluster
        wrong_points = 0 # Clusters with no nearby clusters
        for j in range(cluster_len): # Compare with every other cluster
            if j == i:
                continue # skip comparing with itself

            distance = gpxpy.geo.haversine_distance(
                cluster_centers[i][0], cluster_centers[i][1],
                cluster_centers[j][0], cluster_centers[j][1]
            )

            # Convert meters to miles and track minimum inter-cluster
distance
            distance_miles = distance / (1.60934 * 1000)
            min_dist = min(min_dist, distance_miles)
            # Count as "nice" if distance is <= 2 Miles
            if distance_miles <= 2:
                nice_points += 1
            else:
                wrong_points += 1

        less2.append(nice_points)
        more2.append(wrong_points)

    # Store the nice-points data for analysis later
    neighbours.append(less2)

    avg_within_2 = np.mean(less2)
    avg_outside_2 = cluster_len - avg_within_2


    print(f"On choosing a cluster size of {cluster_len}")

    print(f"Avg. Number of Clusters within the vicinity (i.e.
intercluster-distance < 2): {round(avg_within_2, 2)}\

            {round(100 - (avg_outside_2 * 100 / cluster_len), 2)}%")
```

```
    print(f"Avg. Number of Clusters outside the vicinity (i.e.
intercluster-distance > 2): {round(avg_outside_2, 2)}")

    print(f"Min inter-cluster distance =  {round(min_dist, 3)} Gap
{round(min_dist - 0.5 , 2)}")


# Function to apply MiniBatchKMeans clustering on pickup coordinates
def find_clusters(increment, dataframe):
    # Initialise and fit MiniBatchKMeans with an 'i' number of clusters
    kmeans = MiniBatchKMeans(
        n_clusters = increment, # Number of clusters to form
        batch_size = 10000,     # Mini-batch size for faster convergence
        random_state = 42,      # For reproducibility
    ).fit(coords)

    # Add cluster predictions as a new column to the Polars DataFrame
    dataframe = dataframe.with_columns([
        pl.Series(name = 'pickup_cluster', values =
kmeans.predict(coords))
    ])

    # Extract the coordinates of cluster centers
    cluster_centers = kmeans.cluster_centers_
    cluster_len = len(cluster_centers)
       # Number of clusters (should be equal to increment)

    return cluster_centers, cluster_len

# Loop over different values of K (number of clusters)
for increment in range(10, 100, 10): # for k = 10 -> 100 by 10
    cluster_centers, cluster_len = find_clusters(increment, dataframe =
df_all)
    # Evaluate how tightly packed the clusters are (within 2 miles)
    find_min_distance(cluster_centers, cluster_len)
```

Conclusion

```
On choosing a cluster size of 38
Avg. Number of Clusters within the vicinity (i.e. intercluster-distance < 2): 10.63          27.98%
Avg. Number of Clusters outside the vicinity (i.e. intercluster-distance > 2): 27.37
Min inter-cluster distance =  0.36 Gap -0.14
```

```python
# Step 1: Cluster with k=38
k = 38

# Make sure `coords` is a NumPy array of shape (n_samples, 2) with lat and lon
# Example: coords = df_all.select(['pickup_latitude', 'pickup_longitude']).to_numpy()
coords = df_all.select(['pickup_latitude', 'pickup_longitude']).to_numpy()

# Fit MiniBatchKMeans
kmeans = MiniBatchKMeans(n_clusters=k, batch_size=1_000_000, random_state=42).fit(coords)

# Step 2: Extract cluster centers (lat, lon)
cluster_centers = kmeans.cluster_centers_

# Step 3: Create pandas DataFrame from cluster centers
centers_df = pd.DataFrame(cluster_centers, columns=["cluster_lat", "cluster_lon"])
centers_df["pickup_cluster"] = centers_df.index  # cluster id: 0 to 37

# Step 4: Convert to Polars
cluster_centers_pl = pl.from_pandas(centers_df)

# Step 5: Join to your main Polars DataFrame (df_all)
df_all = df_all.join(
    cluster_centers_pl,
    on="pickup_cluster",
    how="left"
)
```



log_trip_times	pickup_cluster	pickup_unix_binned
4.301163	10	1.4516e9
5.17204	20	1.4516e9
3.449638	10	1.4516e9
3.34664	6	1.4516e9
3.331666	4	1.4516e9

Feature Engineering

feature1 = number of pickup at t - 1 time
feature2 = number of pickup at t - 2 time
feature3 = number of pickup at t - 3 time
feature4 = number of pickup at t - 4 time
feature5 = number of pickup at t - 5 time

feature6= latitude of cluster
feature7 = longitude of cluster
feature8 = clusterID

feature9 = weekday
feature10 = exp_avg

Feature11 = time_stamp

```
import pandas as pd

# ——— 0. Make sure df_all is a Polars DataFrame ———
if isinstance(df_all, pd.DataFrame):
   df_all = pl.from_pandas(df_all)
elif not isinstance(df_all, pl.DataFrame):
   df_all = pl.DataFrame(df_all)

# ——— 1. Parse your binned-datetime into a true Datetime ———
df_all = df_all.with_columns(
   pl.col("pickup_binned_datetime")
     .str.replace('"', "")
     .str.to_datetime("%Y-%m-%dT%H:%M:%S.%3f")
     .alias("pickup_dt")
)

# ——— 2. Aggregate to one row per (cluster, time) ———
df_counts = (
   df_all
   .groupby(["pickup_cluster", "pickup_dt"])
   .agg([
       pl.count().alias("target"),
       pl.first("cluster_lat").alias("lat"),
```

```
        pl.first("cluster_lon").alias("lon"),
    ])
    .sort(["pickup_cluster", "pickup_dt"])
)

# ——— 3. Compute lags, weekday, exp-avg, cluster_id **and time_str**
df_features = df_counts.with_columns([
    # lags
    pl.col("target").shift(1).over("pickup_cluster").alias("ft_1"),
    pl.col("target").shift(2).over("pickup_cluster").alias("ft_2"),
    pl.col("target").shift(3).over("pickup_cluster").alias("ft_3"),
    pl.col("target").shift(4).over("pickup_cluster").alias("ft_4"),
    pl.col("target").shift(5).over("pickup_cluster").alias("ft_5"),

    # cluster ID
    pl.col("pickup_cluster").alias("cluster_id"),

    # weekday
    pl.col("pickup_dt").dt.weekday().alias("weekday"),

    # exponential moving average
    pl.col("target")
      .ewm_mean(alpha=0.3)
      .over("pickup_cluster")
      .alias("exp_avg"),

    # **new**: 10-min time bin as "HH:MM"
    pl.col("pickup_dt")
      .dt.truncate("10m")     # round DOWN to nearest 10m
      .dt.strftime("%H:%M")
      .alias("time_str"),
]).drop_nulls()

# ——— 4. Select + reorder final columns (including time_str) ———
df_final = df_features.select([
    "time_str",
    "ft_5","ft_4","ft_3","ft_2","ft_1",
    "Lat","lon", "weekday","exp_avg", "cluster_id","target"
])
```

Step 1: Parse the time column into a Datetime

- "pickup_binned_datetime" might be a **string** like "2022-01-01T12:00:00.000"
- You **clean** it by removing quotes
- Then convert it to an actual **datetime type** so you can later:
    - Group by time
    - Extract weekdays
    - Sort chronologically

Store the parsed datetime in a new column called "pickup_dt"

Step 2: Group the data — count pickups per cluster & time

- **Count** the number of pickups for each cluster (area) and time bin → this becomes your **target variable** (demand)
- Also, keep the lat and lon of the cluster as **features**
- Sort it by cluster and datetime to compute **lag features**

Now you have one row per (pickup_cluster, time_bin).

Step 3: Feature Engineering – Add lags, weekday, and moving average

I used .over("pickup_cluster")

to make sure these features are calculated **per cluster.**

Feature	What it means	Why it helps
`ft_1` to `ft_5`	Demand 1 to 5 time steps before	Helps model learn patterns & trends
`weekday`	Day of week (0=Mon...6=Sun)	Demand varies by weekday (e.g., weekends)
`exp_avg`	Smoothed average demand	Stabilizes noisy fluctuations
`cluster_id`	Area index	Lets the model learn spatial behavior

ft_5	ft_4	ft_3	ft_2	ft_1	lat	lon	weekday	exp_avg	cluster_id	target
142	250	279	294	318	40.744239	-74.003927	4	293.504945	0	317
250	279	294	318	317	40.744239	-74.003927	4	288.436013	0	278
279	294	318	317	278	40.744239	-74.003927	4	275.24476	0	247
294	318	317	278	247	40.744239	-74.003927	4	266.727635	0	248
318	317	278	247	248	40.744239	-74.003927	4	263.724514	0	257

Export the dataset with these many features.

Dimension 3.25 Million by 11 Features

Machine Learning

Load the Train and Test Datasets.

```
data = pl.read_csv('train.csv')
data_test = pl.read_csv('test.csv')
```

```
X_train = data[['ft_1', 'ft_2', 'ft_3', 'ft_4', 'ft_5', 'lat', 'lon', 'weekday', 'exp_avg']]
y_train = data['target']
```

```
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestRegressor

param_grid = {
    'n_estimators': [10, 200],
    'max_depth': [10, 15, 20, 25, None],
    'min_samples_split': [1, 10],
    'min_samples_leaf': [1, 10],
    'max_features': ['auto', 'sqrt']
}

model = RandomForestRegressor(random_state=42)
grid_search = GridSearchCV(estimator=model,
                           param_grid=param_grid,
                           cv=5, scoring='r2', n_jobs=-1, verbose=2)

grid_search.fit(X_train, y_train)
print("Best params:", grid_search.best_params_)
print("Best R2 score:", grid_search.best_score_)
```

Testing / Result

Input / Output

ft_1	ft_2	ft_3	ft_4	ft_5	lat	lon	week day	Exp_Avg	Target
209	196	240	225	193	40.763863	-73.964451	6	206	205
186	171	205	209	196	40.763863	-73.964451	5	185	194

GitHub - Project:

https://github.com/aydiegithub/nyc-taxi-demand-prediction.git